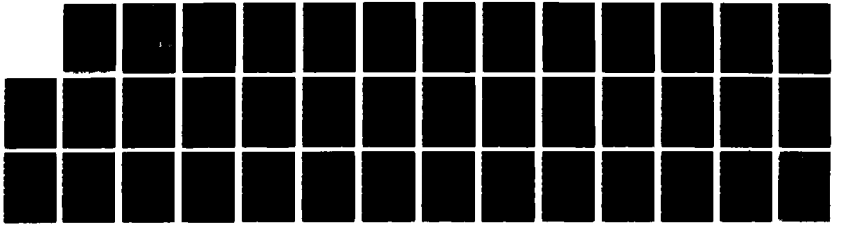
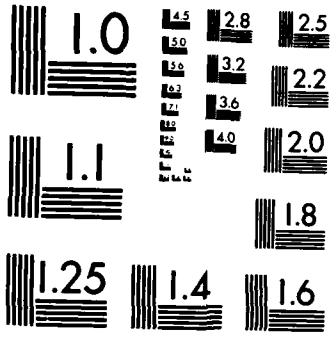
AD-A189 795 ADA (TRADE NAME) COMPILER VALIDATION SUMMARY REPORT: 1/1
HARRIS CORPORATION H (U) INFORMATION SYSTEMS AND
TECHNOLOGY CENTER W-P AFB OH ADA VALI 03 JUN 87
UNCLASSIFIED AVF-VSR-80 0787 F/G 12/5 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

SECURI

DTIC FILE COPY

(2)

AD-A189 795

PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. RE	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER

4. TITLE *(and Subtitle)*
Ada Compiler Validation Summary Report:
Harris Corp., Harris Ada Compiler, Ver. 3.1, Harris H1200

5. TYPE OF REPORT & PERIOD COVERED
3 June 1987 to 3 June 1988

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Wright-Patterson AFB

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION AND ADDRESS
Ada Validation Facility
ASD/SIOL
Wright-Patterson AFB OH 45433-6503

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Ada Joint Program Office
United States Department of Defense
Washington, DC 20301-3081

12. REPORT DATE
33 June 1987

13. NUMBER OF PAGES
37

14. MONITORING AGENCY NAME & ADDRESS*(If different from Controlling Office)*
Wright-Patterson

15. SECURITY CLASS *(of this report)*
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20. If different from Report)*

UNCLASSIFIED

DTIC ELECTE JAN 0 6 1988 S D

18. SUPPLEMENTARY NOTES

19. KEYWORDS *(Continue on reverse side if necessary and identify by block number)*

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

See Attached

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

EXECUTIVE SUMMARY

This Validation Summary Report (VSR) summarizes the results and conclusions of validation testing performed on the Harris Ada Compiler, Version 3.1, using Version 1.8 of the Ada® Compiler Validation Capability (ACVC). The Harris Ada Compiler is hosted on a Harris H1200 operating under VOS, Version 6.1. Programs processed by this compiler may be executed on a Harris H1200 operating under VOS, Version 6.1.

On-site testing was performed 30 May 1987 through 3 June 1987 at Fort Lauderdale, FL, under the direction of the Ada Validation Facility (AVF), according to Ada Validation Organization (AVO) policies and procedures. The AVF identified 2138 of the 2399 tests in ACVC Version 1.8 to be processed during on-site testing of the compiler. The 19 tests withdrawn at the time of validation testing, as well as the 242 executable tests that make use of floating-point precision exceeding that supported by the implementation, were not processed. After the 2138 tests were processed, results for Class A, C, D, and E tests were examined for correct execution. Compilation listings for Class B tests were analyzed for correct diagnosis of syntax and semantic errors. Compilation and link results of Class L tests were analyzed for correct detection of errors. There were 33 of the processed tests determined to be inapplicable. The remaining 2105 tests were passed.

The results of validation are summarized in the following table:

RESULT	CHAPTER												TOTAL
	2	3	4	5	6	7	8	9	10	11	12	14	
Passed	96	219	298	241	161	97	135	261	130	32	218	217	2105
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	20	106	122	6	0	0	4	1	0	0	0	16	275
Withdrawn	0	5	5	0	0	1	1	2	4	0	1	0	19
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399

The AVF concludes that these results demonstrate acceptable conformity to ANSI/MIL-STD-1815A Ada.

®Ada is a registered trademark of the United States Government (Ada Joint Program Office).

AVF Control Number: AVF-VSR-80.0787
87-01-07-HAR

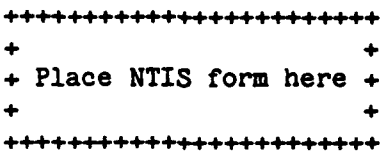
Ada® COMPILER
VALIDATION SUMMARY REPORT:
Harris Corporation
Harris Ada Compiler, Version 3.1
Harris H1200

Completion of On-Site Testing:
3 June 1987

Prepared By:
Ada Validation Facility
ASD/SCOL
Wright-Patterson AFB OH 45433-6503

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington, D.C.

®Ada is a registered trademark of the United States Government (Ada Joint Program Office).


Place NTIS form here

Ada® Compiler Validation Summary Report:

Compiler Name: Harris Ada Compiler, Version 3.1

Host:	Target:
Harris H1200 under VOS, Version 6.1	Harris H1200 under VOS, Version 6.1

Testing Completed 3 June 1987 Using ACVC 1.8

This report has been reviewed and is approved.

Georgeanne C Chitwood

Ada Validation Facility
Georgeanne Chitwood
ASD/SCOL
Wright-Patterson AFB OH 45433-6503

John F Kramer

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA

Virginia L. Castor

Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
Washington DC

®Ada is a registered trademark of the United States Government (Ada Joint Program Office).

EXECUTIVE SUMMARY

This Validation Summary Report (VSR) summarizes the results and conclusions of validation testing performed on the Harris Ada Compiler, Version 3.1, using Version 1.8 of the Ada® Compiler Validation Capability (ACVC). The Harris Ada Compiler is hosted on a Harris H1200 operating under VOS, Version 6.1. Programs processed by this compiler may be executed on a Harris H1200 operating under VOS, Version 6.1.

On-site testing was performed 30 May 1987 through 3 June 1987 at Fort Lauderdale, FL, under the direction of the Ada Validation Facility (AVF), according to Ada Validation Organization (AVO) policies and procedures. The AVF identified 2138 of the 2399 tests in ACVC Version 1.8 to be processed during on-site testing of the compiler. The 19 tests withdrawn at the time of validation testing, as well as the 242 executable tests that make use of floating-point precision exceeding that supported by the implementation, were not processed. After the 2138 tests were processed, results for Class A, C, D, and E tests were examined for correct execution. Compilation listings for Class B tests were analyzed for correct diagnosis of syntax and semantic errors. Compilation and link results of Class L tests were analyzed for correct detection of errors. There were 33 of the processed tests determined to be inapplicable. The remaining 2105 tests were passed.

The results of validation are summarized in the following table:

RESULT	CHAPTER												TOTAL
	2	3	4	5	6	7	8	9	10	11	12	14	
Passed	96	219	298	241	161	97	135	261	130	32	218	217	2105
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	20	106	122	6	0	0	4	1	0	0	0	16	275
Withdrawn	0	5	5	0	0	1	1	2	4	0	1	0	19
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399

The AVF concludes that these results demonstrate acceptable conformity to ANSI/MIL-STD-1815A Ada.

®Ada is a registered trademark of the United States Government (Ada Joint Program Office).

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from characteristics of particular operating systems, hardware, or implementation strategies. All of the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

INTRODUCTION

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any unsupported language constructs required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc., under the direction of the AVF according to policies and procedures established by the Ada Validation Organization (AVO). On-site testing was conducted from 30 May 1987 through 3 June 1987 at Fort Lauderdale, FL.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

Ada Validation Facility
ASD/SCOL
Wright-Patterson AFB OH 45433-6503

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, FEB 1983.

2. Ada Validation Organization: Procedures and Guidelines, Ada Joint Program Office, 1 JAN 1987.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., DEC 1984.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. A set of programs that evaluates the conformity of a compiler to the Ada language specification, ANSI/MIL-STD-1815A.
Ada Standard	ANSI/MIL-STD-1815A, February 1983.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. In the context of this report, the AVF is responsible for conducting compiler validations according to established policies and procedures.
AVO	The Ada Validation Organization. In the context of this report, the AVO is responsible for setting procedures for compiler validations.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	A test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.

Inapplicable test	A test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	A test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	A program that checks a compiler's conformity regarding a particular feature or features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	A test found to be incorrect and not used to check conformity to the Ada language specification. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. However, no checks are performed during execution to see if the test objective has been met. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers

permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of these units is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: Harris Ada Compiler, Version 3.1

ACVC Version: 1.8

Certificate Number: 870601W1.08067

Host Computer:

Machine:	Harris H1200
Operating System:	VOS Version 6.1
Memory Size:	12 megabytes

Target Computer:

Machine:	Harris H1200
Operating System:	VOS Version 6.1
Memory Size:	12 megabytes

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. This compiler is characterized by the following interpretations of the Ada Standard:

- Capacities.

  The compiler correctly processes tests containing loop statements nested to 65 levels, block statements nested to 65 levels, and recursive procedures separately compiled as subunits nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See tests D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

- Universal integer calculations.

  An implementation is allowed to reject universal integer calculations having values that exceed SYSTEM.MAX_INT. This implementation does not reject such calculations and processes them correctly. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

- Predefined types.

  This implementation does not support any additional predefined types in the package STANDARD. (See tests B86001C and B86001D.)

- Based literals.

  An implementation is allowed to reject a based literal with a value exceeding SYSTEM.MAX_INT during compilation, or it may raise NUMERIC_ERROR or CONSTRAINT_ERROR during execution. This implementation raises NUMERIC_ERROR during execution. (See test E24101A.)

- Array types.

  An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT.

  A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises NUMERIC_ERROR when the array type is declared. (See test C52103X.)

A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the array subtype is declared. (See test C52104Y.)

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR when the array type is declared. (See test E52103Y.)

In assigning one-dimensional array types, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Discriminated types.

  During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications. (See test E38104A.)

  In assigning record types with discriminants, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Aggregates.

  In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index type. (See tests C43207A and C43207B.)

  In the evaluation of an aggregate containing subaggregates, all choices are evaluated before being checked for identical bounds. (See test E43212B.)

  All choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

. Functions.

An implementation may allow the declaration of a parameterless function and an enumeration literal having the same profile in the same immediate scope, or it may reject the function declaration. If it accepts the function declaration, the use of the enumeration literal's identifier denotes the function. This implementation rejects the declaration. (See test E66001D.)

. Representation clauses.

The Ada Standard does not require an implementation to support representation clauses. If a representation clause is not supported, then the implementation must reject it. While the operation of representation clauses is not checked by Version 1.8 of the ACVC, they are used in testing other language features. This implementation accepts 'STORAGE_SIZE for tasks, and 'STORAGE_SIZE for collections; it rejects 'SIZE and 'SMALL clauses. Enumeration representation clauses, including those that specify noncontiguous values, appear to be supported. (See tests C55B16A, C87B62A, C87B62B, C87B62C, and BC1002A.)

. Pragmas.

The pragma INLINE is supported for procedures and functions. (See tests CA3004E and CA3004F.)

. Input/output.

The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants. The package DIRECT_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, AE2101H, CE2201D, CE2201E, and CE2401D.)

An existing text file can be opened in OUT_FILE mode, cannot be created in OUT_FILE mode, and cannot be created in IN_FILE mode. (See test EE3102C.)

Only one internal file can be associated with each external file for text I/O. (See tests CE3111A..E (5 tests).)

Only one internal file can be associated with each external file for sequential I/O. (See tests CE2107A..F (6 tests).)

Only one internal file can be associated with each external file for direct I/O. (See tests CE2107A..F (6 tests).)

Temporary sequential files are given a name. Temporary direct files are given a name. Temporary files given names are deleted when they are closed. (See tests CE2108A and CE2108C.)

- Generics.

  Generic subprogram declarations and bodies can be compiled in separate compilations. (See test CA2009F.)

  Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C and BC3205D.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.8 of the ACVC contains 2399 tests. When validation testing of the Harris Ada Compiler was performed, 19 tests had been withdrawn. The remaining 2380 tests were potentially applicable to this validation. The AVF determined that 275 tests were inapplicable to this implementation, and that the 2105 applicable tests were passed by the implementation.

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	69	862	1098	17	13	46	2105
Failed	0	0	0	0	0	0	0
Inapplicable	0	5	270	0	0	0	275
Withdrawn	0	7	12	0	0	0	19
TOTAL	69	874	1380	17	13	46	2399

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER												TOTAL
	2	3	4	5	6	7	8	9	10	11	12	14	
Passed	96	219	298	241	161	97	135	261	130	32	218	217	2105
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	20	106	122	6	0	0	4	1	0	0	0	16	275
Withdrawn	0	5	5	0	0	1	1	2	4	0	1	0	19
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399

3.4 WITHDRAWN TESTS

The following 19 tests were withdrawn from ACVC Version 1.8 at the time of this validation:

C32114A	C41404A	B74101B
B33203C	B45116A	C87B50A
C34018A	C48008A	C92005A
C35904A	B49006A	C940ACA
B37401A	B4A010C	CA3005A..D (4 tests)
		BC3204C

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. For this validation attempt, 275 tests were inapplicable for the reasons indicated:

- C34001D, B52004E, B55B09D, and C55B07B use SHORT_INTEGER which is not supported by this compiler.
- C34001E, B52004D, B55B09C, and C55B07A use LONG_INTEGER which is not supported by this compiler.
- C34001F and C35702A use SHORT_FLOAT which is not supported by this compiler.

- C34001G and C35702B use LONG_FLOAT which is not supported by this compiler.

- B86001D requires a predefined numeric type other than those defined by the Ada language in package STANDARD. There is no such type for this implementation.

- C86001F redefines package SYSTEM, but TEXT_IO is made obsolete by this new definition in this implementation and the test cannot be executed since the package REPORT is dependent on the package TEXT_IO.

- C87B62A and C87B62C use length clauses which are not supported by this compiler. The length clauses are rejected during compilation.

- C96005B checks implementations for which the smallest and largest values in type DURATION are different from the smallest and largest values in DURATION's base type. This is not the case for this implementation.

- CE2107A..F (6 tests), CE2110B, CE2111D, CE2111H, CE3111A..E (5 tests), CE3114B, and CE3115A are inapplicable because multiple internal files cannot be associated with the same external file. The proper exception is raised when multiple access is attempted.

- The following 242 tests require a floating-point accuracy that exceeds the maximum of nine supported by the implementation:

C24113F..Y (20 tests)	C35705F..Y (20 tests)
C35706F..Y (20 tests)	C35707F..Y (20 tests)
C35708F..Y (20 tests)	C35802F..Y (20 tests)
C45241F..Y (20 tests)	C45321F..Y (20 tests)
C45421F..Y (20 tests)	C45424F..Y (20 tests)
C45521F..Z (21 tests)	C45621F..Z (21 tests)

3.6 SPLIT TESTS

If one or more errors do not appear to have been detected in a Class B test because of compiler error recovery, then the test is split into a set of smaller tests that contain the undetected errors. These splits are then compiled and examined. The splitting process continues until all errors are detected by the compiler or until there is exactly one error per split. Any Class A, Class C, or Class E test that cannot be compiled and executed because of its size is split into a set of smaller subtests that can be processed.

Splits were required for 18 Class B tests:

B24204A	B33301A	B67001A
B24204B	B37201A	B67001B
B24204C	B38008A	B67001C
B2A003A	B41202A	B67001D
B2A003B	B44001A	B91003B
B2A003C	B64001A	B95001A

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.8 produced by the Harris Ada Compiler was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and that the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the Harris Ada Compiler using ACVC Version 1.8 was conducted on-site by a validation team from the AVF. The configuration consisted of a Harris H1200 operating under VOS, Version 6.1.

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring splits during the prevalidation testing were included in their split form on the magnetic tape.

The contents of the magnetic tape were loaded directly onto the host computer. After the test files were loaded to disk, the full set of tests was compiled on the Harris H1200, and all executable tests were linked and run. Results were transferred to an HCX-7 via FTP and printed.

The compiler was tested using command scripts provided by Harris Corporation and reviewed by the validation team. The following options were in effect for testing:

Option	Effect
-w	warnings suppressed (all but b tests)
-el	long error listing (for compile-error tests)
-e	short error listing (for executable tests)
-Bf front-end	specify fe
-Bc code-generator	specify cg
-Bl linker	specify a.ld
-Be error-program	specify a.error
-o executable name	specify executable

Tests were compiled, linked, and executed (as appropriate) using a single computer. Test output and compilation listings were captured on magnetic tape and archived at the AVF. The listings and job logs examined on-site by the validation team were also archived.

3.7.3 Test Site

The validation team arrived at Fort Lauderdale, FL on 30 May 1987, and departed after testing was completed on 3 June 1987.

APPENDIX A

DECLARATION OF CONFORMANCE

Harris Corporation has submitted the following declaration of conformance concerning the Harris Ada Compiler.

DECLARATION OF CONFORMANCE

Compiler Implementor: Harris Corporation
Ada® Validation Facility: ASD/SCOL, Wright-Patterson AFB, OH
Ada Compiler Validation Capability (ACVC) Version: 1.8

Base Configuration

Base Compiler Name: Harris Ada Compiler Version: Version 3.1
Host Architecture ISA: Harris H1200 OS&VER #: VOS, Version 6.1
Target Architecture ISA: Harris H1200 OS&VER #: VOS, Version 6.1

Implementor's Declaration

I, the undersigned, representing Harris Corporation, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler listed in this declaration. I declare that Harris Corporation is the owner of record of the Ada language compiler listed above and, as such, is responsible for maintaining said compiler in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for the Ada language compiler listed in this declaration shall be made only in the owner's corporate name.

Wendell Norton Date: 6-2-87
Harris Corporation
Wendell E. Norton, Director of Contracts

Owner's Declaration

I, the undersigned, representing Harris Corporation, take full responsibility for implementation and maintenance of the Ada compiler listed above, and agree to the public disclosure of the final Validation Summary Report. I further agree to continue to comply with the Ada trademark policy, as defined by the Ada Joint Program Office. I declare that the Ada language compilers listed, and its host/target performance are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A. I have reviewed the Validation Summary Report for the compiler and concur with the contents.

Wendell Norton Date: 6-2-87
Harris Corporation
Wendell E. Norton, Director of Contracts

®Ada is a registered trademark of the United States Government (Ada Joint Program Office).

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of MIL-STD-1815A, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the Harris Ada Compiler, Version 3.1, are described in the following sections which discuss topics in Appendix F of the Ada Language Reference Manual (ANSI/MIL-STD-1815A). Implementation-specific portions of the package STANDARD are also included in this appendix.

```
package STANDARD is

     ...

     type INTEGER is range -8388608 .. 8388607;

     type FLOAT is digits 9 range
          -2#0.11111111111111111111111111111111111111#E127 ..
           2#0.11111111111111111111111111111111111111#E127;

     type DURATION is delta 2.0**(-30) range -1.07374182E9 ..
                                              1.07374182E9;

     ...

end STANDARD;
```

Appendix F of the Reference Manual for the Ada® Programming Language

Harris Corporation
Computer Systems Division
Software Development

1. Program Structure and Compilation

A 'main' program must be a non-generic subprogram that is either a procedure or a function returning a STANDARD.INTEGER (the predefined type). A 'main' program may not be an instantiation of a generic subprogram.

2. Pragmas

Implementation-Dependent Pragmas

PRAGMA CONTROLLED is recognized by the implementation but has no effect in this release.

PRAGMA INLINE is implemented as described in Chapter 6.3.2 and Appendix B of the RM. This implementation expands recursive subprograms marked with the pragma up to a maximum nesting depth of 4. Warnings are produced for nesting depths greater than this or for bodies that are not available for inline expansion.

PRAGMA INTERFACE is recognized by the implementation and supports calls to C and FORTRAN language functions. The Ada language specifications can be either functions or procedures. All parameters must have mode IN.

For C, the types of parameters and the result type for functions must be scalar, access, or the predefined type ADDRESS defined in the package SYSTEM. Record and array objects may be passed by reference using the ADDRESS attribute.

The default link name is the symbolic representation of the simple name converted to lower case. The link name of interface routines may be changed via the implementation-defined pragma external_name.

For FORTRAN, all parameters are passed by reference; the parameter types must have the type ADDRESS defined in the package SYSTEM. The result type for a FORTRAN function must be a scalar type. Care should be taken when using tasking and FORTRAN functions. Since FORTRAN is not reentrant we suggest that an Ada controller task should be used to access FORTRAN functions.

The default link name is the symbolic representation of the simple name converted to upper case. The link name of interface routines may be changed via the implementation-defined pragma external_name.

PRAGMA MEMORY_SIZE is recognized by the implementation, but has no effect. The implementation does not allow the package SYSTEM to be modified by means of pragmas; however, the same effect can be achieved by recompiling SYSTEM with altered values.

PRAGMA OPTIMIZE is recognized by the implementation but has no effect in this release.

PRAGMA PACK will cause the compiler to choose a non-aligned representation for composite types. In the current release, it will not cause objects to be packed at the bit level.

PRAGMA SHARED is recognized by the implementation but has no effect in this release.

PRAGMA STORAGE_UNIT is recognized by the implementation but has no effect. The implementation does not allow the package SYSTEM to be modified by means of pragmas; however, the same effect can be achieved by recompiling SYSTEM with altered values.

PRAGMA SUPPRESS is recognized by the implemention and applies from the point of occurrence to the end of the innermost enclosing block. The double parameter form of the pragma, with a name of an object, type, or subtype is recognized, but has no effect.

PRAGMA SYSTEM_NAME is recognized by the implementation but has no effect. The implementation does not allow the package SYSTEM to be modified by means of pragmas; however, the same effect can be achieved by recompiling SYSTEM with altered values.

Implementation-Defined Pragmas

PRAGMA EXTERNAL_NAME allows variables defined in Ada language source code to be referenced from foreign languages. PRAGMA EXTERNAL_NAME will replace all occurences of variable_name with an external reference to link_name in the object file using the format shown below.

```
pragma EXTERNAL_NAME(variable_name, "link_name");
```

This pragma is allowed at the place of a declarative iten in a package specification and must apply to an object declared earlier in the same package specification. The object must be declared as a scalar or an access type. The object cannot be any of the following:

a loop variable
a constant,
an initialized variable,
an array, or
a record.

PRAGMA INTERFACE_OBJECT allows a user to reference an object defined in another language in much the same way that the INTERFACE pragma allows a user to call a subprogram defined in another language. The form of this pragma is:

```
pragma INTERFACE_OBJECT(object_name, "linker_name");
```

PRAGMA INTERFACE_OBJECT provides an interface to objects defined in foreign languages. This pragma has a required first parameter which is the simple name of an Ada variable to be associated with the foreign object. The optional second parameter is a string constant which defines the link name of the object. By default, the link name of the object is the symbolic representation of the simple name converted to lower case. The variable declaration must occur before the pragma and both must occur within the same declarative part or package specification.

PRAGMA INTERFACE_COMMON_OBJECT provides an interface to objects defined in foreign languages as common blocks. Its semantics and syntax are identical to those of pragma interface_object except that the second parameter is required. The second parameter must be a string constant representing the link name of the common block as defined by the foreign language.

PRAGMA INTERFACE_MCOM_OBJECT provides an interface to monitor common objects as defined externally – through a foreign language or some other means. Its semantics and syntax are identical to those of pragma interface_common_object except for an additional third parameter. The required third parameter must be a string constant representing the name of the monitor common disc area. This may be a fully rooted or relative path name or a VOS areaname (Note: the VOS convention of automatically inserting the characters "M " before the final name is retained, therefore, the user should not specify these characters). Specification of this pragma will not cause the monitor common disc area to be created.

PRAGMA SHARED_PACKAGE provides for the sharing and communication of library level packages. All variables declared in a package marked pragma shared_package (henceforth referred to as a shared package) are allocated in a VOS monitor common area which is created and maintained by the implementation. The pragma may only be applied to library level package specifications. Each package specification nested in a shared package will also be shared and all objects declared in the nested packages will reside in the same VOS monitor common area as the outer package.

The implementation restricts the kinds of objects that may be declared in a shared package. No unconstrained or dynamically sized objects may be declared in a shared package. No access type objects may be declared in a shared package. No explicit initialization of objects may occur in a shared package. If any of these restrictions are violated, a warning message will be issued and the package will not be *shared*. These restrictions apply to nested packages as well. Note that if a nested package violates one of the above restrictions, this prevents the sharing of all enclosing packages as well.

Pragma shared_package accepts an optional argument which, if specified, must be a string constant containing a blank separated list of VOS disc area control options as defined by the following:

N=name, which specifies the name of the monitor common disc area to be created. If this parameter is not specified, the implementation will choose a VUE path name and create the file under the .mcom HAPSE subdirectory on the host system.

P=n, where n is the pack number used when the monitor common disc area is created.

G=n, where n is the granule size in sectors used when the monitor common disc area is created.

NR, which specifies that the monitor common area is to be Non-Resident, which is the default.

RS, which specified that the monitor common area is to be ReSident.

L=address, which specifies an address where the monitor common disc area is to be bound into physical memory. This is useful for sharing packages across systems configured with shared memory. Note that the RS control option must be specified if L=address is used.

With the valid application of pragma *shared_package* to a library level package, the following assumptions can be made about the objects declared in the package:

The lifetime of such objects is greater than the lifetime defined by the complete execution of a single program.

The state of such objects is not changed between invocations of programs which reference

the objects; except as defined by the recreation of such programs.

A program which causes the creation of such an object during the elaboration of a *shared package* retains the state of the object, if it previously existed; except as defined by the recreation of such a program.

Programs which attempt to reference the contents of objects declared in shared packages that have not been implicitly or explicitly initialized are technically erroneous as defined by the RM (3.2.1 (18)). However, this implementation does not prevent such references and, in fact, expects them.

Since packages that contain objects that are initialized are not candidates for pragma *shared_package* the implementation suggests that programs be created for the sole purpose of initializing objects in the shared package.

PRAGMA SHARE_BODY is used to indicate a desire to share or not share an instantiation. The pragma may reference the generic unit or the instantiated unit. When it references a generic unit, it sets sharing on/off for all instantiations of that generic, unless overridden by specific SHARE_BODY pragmas for individual instantiations. When it references an instantiated unit, sharing is on/off only for that unit. The default is to share all generics that can be shared, unless the unit uses PRAGMA IN_LINE.

PRAGMA SHARE_BODY is only allowed in the following places: immediately within a declarative part, immediately within a package specification, or after a library unit in a compilation, but before any subsequent compilation unit. The form of this pragma is:

pragma SHARE_BODY (generic_name, boolean_literal)

Note that a parent instantiation is independent of any individual instantiation, therefore recompilation of a generic with different parameters has no effect on other compilations that reference it. The unit that caused compilation of a parent instantiation need not be referenced in any way by subsequent units that share the parent instantiation.

Sharing generics causes a slight execution time penalty because all type attributes must be indirectly referenced (as if an extra calling argument were added). However, it substantially reduces compilation time in most circumstances and reduces program size.

3. Implementation-Dependent Attributes

HAPSE has defined the following two attributes for use in conjunction with the implementation-defined pragma shared_package.

P'LOCK for a prefix P which denotes a package.
P'UNLOCK for a prefix P which denotes a package.

These attributes are only applicable to packages that have had pragma shared package applied to them. The 'LOCK attribute defines a function which alters the "state" of the package to the LOCK state. The function has two optional parameters and returns a BOOLEAN result that has the value TRUE if a successful LOCK operation occurred, or FALSE if the package was already LOCKed. The 'UNLOCK attribute defines a function which alters the "state" of the package to the UNLOCK state. It has no parameters and returns a BOOLEAN result that has the value TRUE if a successful UNLOCK operation occurred, or FALSE if the package was already UNLOCKed.

The "state" of the package is only meaningful to the 'LOCK and 'UNLOCK attribute functions, which set and query the state. A LOCK state does not prevent concurrent access to objects in the shared package. These attributes only provide indivisible operations for the set and test of implicit semaphores which could be used to control access.

The first parameter of the 'LOCK attribute function must be of the base type BOOLEAN and specifies whether to put the program into a sleep state until such time as the package becomes UNLOCKed, before executing the LOCK operation. This parameter is declared with a default value of FALSE, such that no sleep will occur unless explicitly specified by the user. The sleep state is induced through the VOS $WAIT service. Note that the sleep state will not be pre-empted by the implementation's time-slice for tasks. Note that even if sleep is requested, this does not guarentee that the LOCK operation will be successful when it finally is attempted.

The second parameter must be of the base type INTEGER and represents the timeout period in clock ticks, should the function be requested to sleep. This parameter defaults to zero which requests no timeout.

4. Specification of the package SYSTEM

```
package SYSTEM is

   type ADDRESS is private  ;
   type NAME is ( harris_vue ) ;

   SYSTEM_NAME              : constant NAME := harris_vue ;

   -- System-Dependent Constraints

   STORAGE_UNIT             : constant := 8 ;
   MEMORY_SIZE              : constant := 6_291_456 ;

   -- System-Dependent Named Numbers

   MIN_INT                  : constant := - 8_388_608 ;
   MAX_INT                  : constant := 8_388_607 ;
   MAX_DIGITS               : constant := 9 ;
   MAX_MANTISSA             : constant := 31 ;
   FINE_DELTA               : constant := 2.0**(-30) ;
   TICK                     : constant := 0.01 ;

   -- Other System-dependent Declarations

   subtype PRIORITY is INTEGER range 0 .. 23 ;

   MAX_REC_SIZE : integer := 32_767 * 3 ;

private

   type ADDRESS is new INTEGER;

end SYSTEM ;
```

5. Restrictions on Representation Clauses

Pragma PACK

Bit packing is not supported. In the presence of pragma PACK, components of composite types are packed to the nearest whole STORAGE_UNIT.

Length Clauses

The specification T'SIZE is not supported. The specification T'SMALL is not supported.

Record Representation Clauses

Component clauses must specify alignment on multiples of 3 STORAGE_UNIT boundaries.

Address Clauses

Address clauses and interrupts are not supported.

6. Other Representation Implementation-Dependencies

Change of representation is not supported for record types.

The ADDRESS attribute is not supported for the following entities: static constants; packages; tasks; labels; and entries.

Machine code insertions are not supported.

7. Conventions for Implementation-Generated Names

There are no implementation generated names.

8. Interpretation of Expressions in Address Clauses

Address clauses and interrupts are not supported.

9. Restrictions on Unchecked Conversions

The predefined generic function UNCHECKED_CONVERSION cannot be instantiated with a target type that is an unconstrained array type or an unconstrained record type with discriminants.

10. Implementation Characteristics of I/O Packages

Interpretation of Strings as Applied to External Files

Strings that contain names of external files are interpreted in the following manners for each of the respective external file environments.

VUE external files: file names may be composed of up to 512 characters of the ASCII character set except for "/", ascii.nul, and non-printable characters. Further, the first character of a file must be alpha-numeric, "." or "_". If the "/" character is encountered in a string, it is interpreted as a separater between file names that specify VUE directories.

VOS external files: file names are composed of a 1 to 8 character qualifier plus a 1 to 8 character areaname. The first character of the areaname must be alphabetic. The remaining characters comprising the areaname may be drawn from the following set of characters: A-Z, 0-9, :, #, -, /, . and " ". The qualifier portion of a file name is optional. If specified, it must be comprised of an account portion, a name portion, and an asterisk. The account portion may be null, or 1-4 characters from the following set: 0-9. The name portion may be null, or 1-4 characters from the following set: A-Z, 0-9. The name portion may not be null if the account portion is not null. If lower case letters are encountered in the string they are converted to upper case.

Interpretation of Strings as Applied to Form Parameters

The OPEN and CREATE I/O procedures accept FORM parameters, in order to specify implementation dependent attributes of files. The HAPSE implementation supports the attributes described below. These attributes may be specified in any order. Blanks may be inserted between attributes, however none are required. No attribute can be specified more than once. All attributes must be specified in uppercase. These attributes are only applicable to CREATE calls. A form string passed to OPEN is ignored.

File Type Attributes

BL	Blocked file
UB	Unblocked file
RA	Random file

These attributes specify the VOS file type of a file to be created. UB is the default for all files. In general, the defaults should not be overrideen for direct and sequential I/O.

Double Buffered Blocking

DB	Defines a BL type file as permanently double buffered

This attribute can only be specified if the file type is BL.

Directory Type

CD	The VOS directory entry for this file is to be kept resident
DD	The VOS directory entry is kept on disc

Access Parameters

PR	PUBLIC READ
PW	PUBLIC WRITE
PD	PUBLIC DELETE
AR	ACCOUNT READ
AW	ACCOUNT WRITE
AD	ACCOUNT DELETE
OW	OWNER WRITE
OD	OWNER DELETE

These attributes determine the access permissions associated with a file. The default access is OW OD. Note that if any access attributes are specified, then only the specified accesses will be granted (i.e. OW OD is not assumed).

File Definition Attributes

A=n	Access level, n = 0-15, VOS access required to access file
B=n	Blocking factor, where n is 1-7 sectors
C=n	Current size, where n is the number of sectors requested
E=n	Eliminate date, where n is the number of days before purging
G=n	Granule size, where n is the number of sectors per granule
M=n	Maximum size, n = number of sectors to which file may expand
P=n	Pack number, n = pack number of pack on which to create file
T=n	Type number, n = 0-7, provided for user file classification

No spaces are allowed between the attribute letter, the equal sign, and the integer value.

Implementation-Dependent Characteristics of DIRECT_IO

Instantiations of DIRECT_IO use the value MAX_REC_SIZE as the record size (expressed in STORAGE_UNITs) when the size of ELEMENT_TYPE exceeds that value. For example, for unconstrained arrays such as string where ELEMENT_TYPE'SIZE is very large, MAX_REC_SIZE is used instead. MAX_REC_SIZE is defined in SYSTEM and can be changed by a program before instantiating DIRECT_IO to provide an upper limit on the record size. In any case, the maximum size supported is 32_768 * 3 * STORAGE_UNIT

bits. DIRECT_IO will raise USE_ERROR if MAX_RECORD_SIZE exceeds this absolute limit.

Implementation-Dependent Characteristics of SEQUENTIAL_IO

Instantiations of SEQUENTIAL_IO use the value MAX_REC_SIZE as the record size (expressed in STORAGE_UNITs) when the size of ELEMENT_TYPE exceeds that value. For example, for unconstrained arrays such as string where ELEMENT_TYPE'SIZE is very large, MAX_REC_SIZE is used instead. MAX_REC_SIZE is defined in SYSTEM and can be changed by a program before instantiating SEQUENTIAL_IO to provide an upper limit on the record size. In any case, the maximum size supported is 32_768 * 3 * STORAGE_UNIT bits. SEQUENTIAL_IO will raise USE_ERROR if MAX_REC_SIZE exceeds this absolute limit.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	(1..498 =>'A', 499 =>'1')
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	(1..498 =>'A', 499 =>'2')
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	(1..249 | 251..499 =>'A', 250 =>'3')
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	(1..249 | 251..499 =>'A', 250 =>'4')
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..496 =>'0', 497..499 =>"298")

Name and Meaning	Value
$BIG_REAL_LIT A real literal that can be either of floating- or fixed-point type, has value 690.0, and has enough leading zeroes to be the size of the maximum line length.	(1..493 =>'0', 494..499 =>"69.0E1"
$BLANKS A sequence of blanks twenty characters fewer than the size of the maximum line length.	(1..479 =>' ')
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	8388607
$EXTENDED_ASCII_CHARS A string literal containing all the ASCII characters with printable graphics that are not in the basic 55 Ada character set.	"abcdefghijklmnopqrstuvwxyz!$%?@[\]^`{}~"
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	8388607
$FILE_NAME_WITH_BAD_CHARS An illegal external file name that either contains invalid characters, or is too long if no invalid characters exist.	./^BAD-CHARACTER
$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character, or is too long if no wild card character exists.	./CE2102{254 C's}
$GREATER_THAN_DURATION A universal real value that lies between DURATION'BASE'LAST and DURATION'LAST if any, otherwise any value in the range of DURATION.	100_000.0
$GREATER_THAN_DURATION_BASE_LAST The universal real value that is greater than DURATION'BASE'LAST, if such a value exists.	10_000_000_000.0

Name and Meaning	Value
$ILLEGAL_EXTERNAL_FILE_NAME1 An illegal external file name.	./^ILLEGAL_EXTERNAL_FILE_NAME1
$ILLEGAL_EXTERNAL_FILE_NAME2 An illegal external file name that is different from $ILLEGAL_EXTERNAL_FILE_NAME1.	/no/such/directory/ILLEGAL_EXT_FILE_NAME2
$INTEGER_FIRST The universal integer literal expression whose value is INTEGER'FIRST.	-8388608
$INTEGER_LAST The universal integer literal expression whose value is INTEGER'LAST.	8388607
$LESS_THAN_DURATION A universal real value that lies between DURATION'BASE'FIRST and DURATION'FIRST if any, otherwise any value in the range of DURATION.	-100_000.0
$LESS_THAN_DURATION_BASE_FIRST The universal real value that is less than DURATION'BASE'FIRST, if such a value exists.	-10_000_000_000.0
$MAX_DIGITS The universal integer literal whose value is the maximum digits supported for floating-point types.	9
$MAX_IN_LEN The universal integer literal whose value is the maximum input line length permitted by the implementation.	499
$MAX_INT The universal integer literal whose value is SYSTEM.MAX_INT.	8388607

TEST PARAMETERS

Name and Meaning	Value
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER if one exists, otherwise any undefined name.	LONG_LONG_INTEGER
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFD#
$NON_ASCII_CHAR_TYPE An enumerated type definition for a character type whose literals are the identifier NON_NULL and all non-ASCII characters with printable graphics.	(NON_NULL)

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 19 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

- C32114A: An unterminated string literal occurs at line 62.
- B33203C: The reserved word "IS" is misspelled at line 45.
- C34018A: The call of function G at line 114 is ambiguous in the presence of implicit conversions.
- C35904A: The elaboration of subtype declarations SFX3 and SFX4 may raise NUMERIC_ERROR instead of CONSTRAINT_ERROR as expected in the test.
- B37401A: The object declarations at lines 126 through 135 follow subprogram bodies declared in the same declarative part.
- C41404A: The values of 'LAST and 'LENGTH are incorrect in the if statements from line 74 to the end of the test.
- B45116A: ARRPRIBL1 and ARRPRIBL2 are initialized with a value of the wrong type--PRIBOOL_TYPE instead of ARRPRIBOOL_TYPE--at line 41.
- C48008A: The assumption that evaluation of default initial values occurs when an exception is raised by an allocator is incorrect according to AI-00397.
- B49006A: Object declarations at lines 41 and 50 are terminated incorrectly with colons, and end case; is missing from line 42.
- B4A010C: The object declaration in line 18 follows a subprogram body of the same declarative part.

WITHDRAWN TESTS

- B74101B: The begin at line 9 causes a declarative part to be treated as a sequence of statements.

- C87B50A: The call of "/=" at line 31 requires a use clause for package A.

- C92005A: The "/=" for type PACK.BIG_INT at line 40 is not visible without a use clause for the package PACK.

- C940ACA: The assumption that allocated task TT1 will run prior to the main program, and thus assign SPYNUMB the value checked for by the main program, is erroneous.

- CA3005A..D (4 tests): No valid elaboration order exists for these tests.

- BC3204C: The body of BC3204C0 is missing.

END
DATE
FILM
4-88
DTIC